

1. 知微 Java 编程规范

1.1. 排版

1.1.1. 规则

1. 程序块要采用缩进风格编写，缩进的空格数为4个，不允许使用TAB缩进。(1.42+)

说明：缩进使程序更易阅读，使用空格缩进可以适应不同操作系统与不同开发工具。

2. 分界符（如大括号‘{’和‘}’）应各独占一行，同时与引用它们的语句左对齐。在函数体的开始、类和接口的定义、以及if、for、do、while、switch、case语句中的程序或者static、synchronized等语句块中都要采用如上的缩进方式。(1.42+)

示例：

```
if (a>b)
{
    doStart();
}
```

3. 较长的语句、表达式或参数（>80字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。(1.42+)

示例：

```
if (logger.isDebugEnabled())
{
    logger.debug("Session destroyed,call-id"
        + event.getSession().getCallId());
}
```

4. 不允许把多个短语句写在一行中，即一行只写一条语句(1.42+)

说明：阅读代码更加清晰

示例：如下例子不符合规范。

```
Object o = new Object(); Object b = null;
```

5. if, for, do, while, case, switch, default 等语句自占一行，且if, for, do, while,switch等语句的执行语句无论多少都要加括号{},case 的执行语句中如果定义变量必须加括号{}。(1.42+)

说明：阅读代码更加清晰，减少错误产生

示例：

```
if (a>b)
{
    doStart();
}
```

```

}

case x:
{
    int i = 9;
}

```

6. 相对独立的程序块之间、变量说明之后必须加空行。(1.42+)

说明:阅读代码更加清晰

示例:

```

if(a > b)
{
    doStart();
}

//此处是空行

return;

```

7. 在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如.），后不应加空格。(1.42+)

说明: 阅读代码更加清晰

示例:

```

if (a == b)
{
    objectA.doStart();
}

a *= 2;

```

1.1.2. 建议

1. 类属性和类方法不要交叉放置，不同存取范围的属性或者方法也尽量不要交叉放置。(1.42+)

格式:

类定义

```

{
    类的公有属性定义
    类的保护属性定义
    类的私有属性定义
    类的公有方法定义

```

```

    类的保护方法定义
    类的私有方法定义
}

```

2. 修饰词按照指定顺序书写: [访问权限][static][final]。(1.42+)

示例:

```
public static final String str = "abc";
```

1.2. 注释

1.2.1. 规则

1. 源程序注释量必须在30%以上。(1.42+)

说明: 由于每个文件的代码注释不一定都可以达到30%, 建议以一个系统内部模块作为单位进行检查

2. 包的注释: 写入一个名为 **package.html** 的**HTML**格式的说明文件放入包所在路径。包的注释内容: 简述本包的作用、详细描述本包的内容、产品模块名称和版本、公司版权。(1.42+)

说明: 方便JavaDoc收集, 方便对包的了解

示例:

```
com/zhiwei/iin/websmap/comm/package.html
```

```

<html>
<body>
<p>一句话简述。
<p>详细描述。
<p>产品模块名称和版本
<br>公司版权信息
</body>
</html>

```

示例:

```

<html>
<body>
<P>为 WEBSMAP 提供通信类, 上层业务使用本包的通信类与 SMP-B 进行通信。
<p>详细描述。。。。。。。。
<p>IIN V100R001 WEBSMAP
<br>(C) 版权所有 2000-2001 知微科技有限公司
</body>
</html>

```

3. 类和接口的注释放在**class** 或者 **interface** 关键字之前, **import** 关键字之后。注释

主要是一句话功能简述与功能详细描述。类注释使用“/ */”注释方式(1.42+)**

说明：方便JavaDoc收集,没有import可放在package之后。注释可根据需要列出：作者、内容、功能、与其它类的关系等。功能详细描述部分说明该类或者接口的功能、作用、使用方法和注意事项，每次修改后增加作者和更新版本号 and 日期，@since 表示从那个版本开始就有这个类或者接口，@deprecated 表示不建议使用该类或者接口。

```
/**
 * <一句话功能简述>
 * <功能详细描述>
 * @author [作者] (必须)
 * @see [相关类/方法] (可选)
 * @since [产品/模块版本] (必须)
 * @deprecated (可选)
 */
```

示例：

```
package com.zhiwei.iin.logwebsmap.comm;

import java.util.*;

/**
 * LogManager 类集中控制对日志读写的操作。
 * 全部为静态变量和静态方法，对外提供统一接口。分配对应日志类型的读写器，
 * 读取或写入符合条件的日志纪录。
 * @author 张三，李四，王五
 * @see LogIteraotor
 * @see BasicLog
 * @since CommonLog1.0
 */
public class LogManager
```

4. 类属性(成员变量)、公有和保护方法注释：写在类属性、公有和保护方法上面，注释方式为“/ */”.(1.42+)**

示例：

```
/**
 * 注释内容
 */
private String logType;

/**
 * 注释内容
 */
public void write()
```

5. 公有和保护方法注释内容：列出方法的一句话功能简述、功能详细描述、输入参数、输出参数、返回值、异常等。(1.42+)

格式:

```
/**
 * <一句话功能简述>
 * <功能详细描述>
 * @param [参数1] [参数1说明]
 * @param [参数2] [参数2说明]
 * @return [返回类型说明]
 * @exception/throws [异常类型] [异常说明]
 * @see [类、类#方法、类#成员]
 * @since [起始版本]
 * @deprecated
 */
```

说明: @since 表示从那个版本开始就有这个方法, 如果是最初版本就存在的方法无需说明;
@exception或throws 列出可能仍出的异常; @deprecated 表示不建议使用该方法。

示例:

```
/**
 * 根据日志类型和时间读取日志。
 * 分配对应日志类型的LogReader, 指定类型、查询时间段、条件和反复器缓冲数,
 * 读取日志记录。查询条件为null或0的表示没有限制, 反复器缓冲数为0读不到日志。
 *
 * 查询时间为左包含原则, 即 [startTime, endTime) 。
 * @param logTypeName 日志类型名 ( 在配置文件中定义的 )
 * @param startTime 查询日志的开始时间
 * @param endTime 查询日志的结束时间
 * @param logLevel 查询日志的级别
 * @param userName 查询该用户的日志
 * @param bufferNum 日志反复器缓冲记录数
 * @return 结果集, 日志反复器
 * @since 1.2
 */
public static LogIterator read(String logType, Date startTime,
Date endTime, int logLevel, String userName, int bufferNum)
```

6. 对于方法内部用**throw**语句抛出的异常, 必须在方法的注释中标明, 对于所调用的其他方法所抛出的异常, 选择主要的在注释中说明。 对于非**RuntimeException**, 即**throws**子句声明会抛出的异常, 必须在方法的注释中标明。(1.42+)

说明: 异常注释用@exception或@throws表示, 在JavaDoc中两者等价, 但推荐用@exception标注Runtime异常, @throws标注非Runtime异常。异常的注释必须说明该异常的含义及什么条件下抛出该异常。

7. 注释应与其描述的代码相近, 对代码的注释应放在其上方, 并与其上面的代码用空行隔开, 注释与所描述内容进行同样的缩排。(1.42+)

说明：可使程序排版整齐，并方便注释的阅读与理解。

示例：

```
/*
 * 注释
 */
public void example2( )
{
    // 注释
    CodeBlock One

    // 注释
    CodeBlock Two
}

/*
 * 注释
 */
public void example( )
{
    // 注释
    CodeBlock One

    // 注释
    CodeBlock Two
}
```

8. 对于switch语句下的case语句，必须在每个case分支结束前加上break语句。(1.42+)

说明：break才能真正表示该switch执行结束，不然可能会进入该case以后的分支。至于语法上合法的场景“一个case后进入下一个case处理”，应该在编码设计上就避免。

9. 修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。(1.42+)

10. 注释的内容要清楚、明了，含义准确，防止注释二义性。(1.42+)

说明：错误的注释不但无益反而有害。

11. 避免在注释中使用缩写，特别是不常用缩写。(1.42+)

说明：在使用缩写时或之前，应对缩写进行必要的说明。

12. 对重载父类的方法必须进行@Override声明(5.0+)

说明：可清楚说明此方法是重载父类的方法，保证重载父类的方法时不会因为单词写错而造成错误(写错方法名或者参数个数，类型都会编译无法通过)

示例：

```
@Override
public void doRequest(SipServletRequest req) throws ServletException,
```

IOException

1.2.2. 建议

1. 避免在一行代码或表达式的中间插入注释。(1.42+)

说明：除非必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

2. 在代码的功能、意图层次上进行注释，提供有用、额外的信息。(1.42+)

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

示例：如下注释意义不大。

```
// 如果 receiveFlag 为真
if (receiveFlag)
```

而如下的注释则给出了额外有用的信息。

```
// 如果从连结收到消息
if (receiveFlag)
```

3. 对关键变量的定义和分支语句（条件分支、循环语句等）必须编写注释。(1.42+)

说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

4. 注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非能用非常流利准确的英文表达。中文注释中需使用中文标点。方法和类描述的第一句话尽量使用简洁明了的话概括一下功能，然后加以句号。接下来的部分可以详细描述。(1.42+)

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文。JavaDoc工具收集简介的时候使用选取第一句话。

5. 方法内的单行注释使用 //。(1.42+)

说明：调试程序的时候可以方便的使用 /* ... */ 注释掉一长段程序。

6. 一些复杂的代码需要说明。(1.42+)

示例：这里主要是对闰年算法的说明。

```
//1. 如果能被4整除，是闰年；
//2. 如果能被100整除，不是闰年；
//3. 如果能被400整除，是闰年。
```

7. 使用Html标签使JavaDoc生成更加美观。(1.42+)

示例：

```
/**
```

```

* Returns a hash code for this string. The hash code for a
* String object is computed as
* 

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```


* using int arithmetic, where s[i] is
the
* ith character of the string, n is the
length * of
* the string, and ^ indicates exponentiation.
* (The hash value of the empty string is zero.)
*
* @return a hash code value for this object.
*/
public int hashCode()

```

生成后的JavaDoc

Returns a hash code for this string. The hash code for a `String` object is computed as

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \dots + s[n-1]$$

using `int` arithmetic, where `s[i]` is the *i*th character of the string, *n* is the length of the string, and `^` indicates exponentiation. (The hash value of the empty string is zero.)

Returns:

a hash code value for this object.

1. 生成后的JavaDoc

1.3. 命名

1.3.1. 规则

1. 类名和接口使用类意义完整的英文描述，每个英文单词的首字母使用大写、其余字母使用小写的大小写混合法。(1.42+)

示例：OrderInformation, CustomerList, LogManager, LogConfig, SmpTransaction

2. 方法名使用类意义完整的英文描述：第一个单词的字母使用小写、剩余单词首字母大写其余字母小写的大小写混合法。(1.42+)

示例：

```
private void calculateRate();
public void addNewOrder();
```

3. 方法中，存取属性的方法采用setter 和 getter方法，动作方法采用动词和动宾结构。(1.42+)

格式：

get + 非布尔属性名()

is + 布尔属性名()

set + 属性名()

动词()

动词 + 宾语()

示例:

```
public String getType();
public boolean isFinished();
public void setVisible(boolean);
public void show();
public void addKeyListener(Listener);
```

4. 属性名使用意义完整的英文描述，第一个单词的字母使用小写，剩余单词首字母大写其余字母小写的大小写混合法。属性名不能与方法名相同。(1.42+)

示例:

```
private customerName;
private orderNumber;
private smpSession;
```

5. 常量名使用全大写的英文描述，英文单词之间用下划线分隔开，并且使用 **static final**修饰。(1.42+)

示例:

```
public static final int MAX_VALUE = 1000;
public static final String DEFAULT_START_DATE = "2001-12-08";
```

1.3.2. 建议

1. 包名采用域后缀倒置的加上自定义的包名，采用小写字母，都应该以com.zhiwei开头(不包括一些特殊原因)。在部门内部应该规划好包名的范围，防止产生冲突。部门内部产品使用部门的名称加上模块名称。产品线的产品使用产品的名称加上模块的名称。(1.42+)

说明：除特殊原因包结构都必须以com.zhiwei开头，如果因为OEM合作等关系，可以不做要求。

格式:

com.zhiwei.产品名.模块名称

示例:

wordcloud包名 com.zhiwei.wordcloud.wcModule

2. 通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的。(1.42+)

说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

3. 常用组件类的命名以组件名加上组件类型名结尾。(1.42+)

示例:

Application 类型的, 命名以App 结尾——MainApp

Frame 类型的, 命名以Frame 结尾——TopoFrame

Panel 类型的, 建议命名以Panel 结尾——CreateCircuitPanel

Bean 类型的, 建议命名以Bean 结尾——DataAccessBean

EJB 类型的, 建议命名以EJB 结尾——DBProxyEJB

Applet 类型的, 建议命名以Applet 结尾——PictureShowApplet

4. 如果函数名超过15 个字母, 可采用以去掉元音字母的方法或者以行业内约定俗成的缩写方式缩写函数名。(1.42+)

示例:

getCustomerInformation() 改为 getCustomerInfo()

5. 准确地确定成员函数的存取控制符号:只是该类内部调用的函数使用 **private** 属性, 继承类可以使用的使用**protected**属性, 同包类可以调用的使用默认属性(不加属性控制符号), 对外公开的函数使用**public**属性(1.42+)

示例:

```
protected void getUserName()
{
    . . . . .
}

private void calculateRate()
{
    . . . . .
}
```

6. 含有集合意义的属性命名, 尽量包含其复数的意义。(1.42+)

示例:

customers, orderItems

1.4. 编码

1.4.1. 规则

1. 数据库操作、IO操作等需要使用结束close()的对象必须在try -catch-finally 的finally中close(), 如果有多个IO对象需要close(), 需要分别对每个对象的close()方法进行try-catch,防止一个IO对象关闭失败其他IO对象都未关闭。(1.42+)

示例:

```
try
```

```

{
    // ... ...
}
catch(IOException ioe)
{
    //... ...
}
finally
{
    try
    {
        out.close();
    }
    catch (IOException ioe)
    {
        //... ...
    }

    try
    {
        in.close();
    }
    catch (IOException ioe)
    {
        //... ...
    }
}

```

2. 系统非正常运行产生的异常捕获后，如果不对该异常进行处理，则应该记录日志。

(1.42+)

说明：此规则指通常的系统非正常运行产生的异常，不包括一些基于异常的设计。若有特殊原因必须用注释加以说明。

示例：

```

try
{
    //..... ...
}
catch (IOException ioe)
{
    logger.error(ioe);
}

```

3. 自己抛出的异常必须要填写详细的描述信息。 **(1.42+)**

说明：便于问题定位。

示例：

```

throw new IOException("Writing data error! Data: " +
data.toString());

```

4. 运行时异常使用**RuntimeException**的子类来表示，不用在可能抛出异常的方法声明上加**throws**子句。非运行期异常是从**Exception**继承而来的，必须在方法声明上加**throws**子

句。(1.42+)

说明:

非运行期异常是由外界运行环境决定异常抛出条件的异常，例如文件操作，可能受权限、磁盘空间大小的影响而失败，这种异常是程序本身无法避免的，需要调用者明确考虑该异常出现时该如何处理方法，因此非运行期异常必须有throws子句标出，不标出或者调用者不捕获该类型异常都会导致编译失败，从而防止程序员本身疏忽。

运行期异常是程序在运行过程中本身考虑不周导致的异常，例如传入错误的参数等。抛出运行期异常的目的是防止异常扩散，导致定位困难。因此在做异常体系设计时要根据错误的性质合理选择自定义异常的继承关系。

还有一种异常是Error继承而来的，这种异常由虚拟机自己维护，表示发生了致命错误，程序无法继续运行例如内存不足。我们自己的程序不应该捕获这种异常，并且也不应该创建该种类型的异常。

5. 在程序中使用异常处理还是使用错误返回码处理，根据是否有利于程序结构来确定，并且异常和错误码不应该混合使用，推荐使用异常。(1.42+)

说明:

一个系统或者模块应该统一规划异常类型和返回码的含义。

但是不能用异常来做一般流程处理的方式，不要过多地使用异常，异常的处理效率比条件分支低，而且异常的跳转流程难以预测。

注意: Java 5.0 程序内部的错误码可以使用枚举来表示。

6. 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。(1.42+)

说明: 防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

示例:

下列语句中的表达式

```
word = (high << 8) | low      (1)
if ((a | b) && (a & c))      (2)
if ((a | b) < (c & d))      (3)
```

如果书写为

```
high << 8 | low
a | b && a & c
a | b < c & d
```

(1) (2) 虽然不会出错，但语句不易理解；(3) 造成了判断条件出错。

7. 避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的静态变量或者枚举来代替。使用异常来表示方法执行错误，而不是使用C++的错误返回码方式。(1.42+)

示例: 如下的程序可读性差。

```
if (state == 0)
{
    state = 1;
    ... // program code
}
```

应改为如下形式:

```
private final static int TRUNK_IDLE = 0;
private final static int TRUNK_BUSY = 1;
private final static int TRUNK_UNKNOWN = -1;

if (state == TRUNK_IDLE)
{
    state = TRUNK_BUSY;
    ... // program code
}
```

注意: Java 5.0 下建议使用枚举来表示。

异常:

```
public void function()
{
    ...
    throw new RuntimeException("。。。");
}
```

8. 数组声明的时候使用 `int[] index` , 而不要使用 `int index[]` 。 (1.42+)

说明: 使用 `int index[]` 格式使程序的可读性较差, `int [] index` 表示声明了一个 `int` 数组(`int []`) 叫做 `index`

示例:

如下程序可读性差:

```
public int getIndex() []
{
    ....
}
```

如下程序可读性好:

```
public int[] getIndex()
{
    ....
}
```

9. 不要使用 `System.out` 与 `System.err` 进行控制台打印, 应该使用工具类(如: 日志工具)进行统一记录或者打印。 (1.42+)

说明: 代码发布的时候可以统一关闭控制台打印, 代码调试的时候又可以打开控制台打印, 方便调试。

10. 用调测开关来切换软件的 **DEBUG** 版和正式版, 而不要同时存在正式版本和 **DEBUG** 版本的不同源文件, 以减少维护的难度。 (1.42+)

11. 集合必须指定模板类型(5.0+)

说明: 方便程序阅读, 除去强制转换代码

示例:

```
Map<String,MyObject> map = new HashMap<String,MyObject>();
```

12. 一个文件不要定义两个类(并非指内部类)。(1.42+)

说明：方便程序的阅读与代码的维护

13. 所有的数据类必须覆写toString()、hashCode()、equals() 方法，toString()方法返回该类有意义的内容。(1.42+)

说明：方便数据类的比较，父类如果实现了比较合理的toString()，子类可以继承不必再重写。hashCode与equals可以使用eclipse自动生成。

示例：

```
public TopoNode
{
    private String nodeName;

    public String toString()
    {
        return "nodeName : " + nodeName;
    }
}
```

14. 判断语句不要使用"* == true"来判断为真

说明：方便阅读，减少没有必要的计算

以下错误：

```
if (ok == true)
{
    .....
}
```

以下正确：

```
if (ok)
{
    .....
}
```

15. 不要写没有必要的向上强制转型。(1.42+)

说明：没必要写的向上强制转型会浪费性能，增加代码阅读难度

示例：

以下错误：

```
FileInputStream fis = new FileInputStream(f);
```

```
InputStream is = (InputStream) fis;
```

1.4.2. 建议

1. 记录异常不要保存`exception.getMessage()`，而要记录`exception.toString()`，一般可通过日志工具记录完整的异常堆栈信息。(1.42+)

说明：NullPointerException抛出时常常描述为空，这样往往看不出是出了什么错。

示例：

```
try
{
    ...
}
catch (FileNotFoundException e)
{
    logger.error(e);
}
```

2. 一个方法不应抛出太多类型的异常。(1.42+)

说明：如果程序中需要分类处理，则将异常根据分类组织成继承关系。如果确实有很多异常类型首先考虑用异常描述来区别，throws/exception子句标明的异常最好不要超过三个。

3. 异常捕获尽量不要直接 `catch (Exception ex)`，应该把异常细分处理。(1.42+)

说明：可以设计更合理异常处理分支

4. 如果多段代码重复做同一件事情，那么在方法的划分上可能存在问题。(1.42+)

说明：若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成为一个新的方法。

5. 集合中的数据如果不使用了应该及时释放，尤其是可重复使用的集合。(1.42+)

说明：由于集合保存了对象的引用，虚拟机的垃圾收集器就不会回收。

6. 源程序中关系较为紧密的代码应尽可能相邻。(1.42+)

说明：便于程序阅读和查找。

示例：矩形的长与宽关系较密切，放在一起。

```
rect.length = 10;
rect.width = 5;
```

7. 不要使用难懂的技巧性很高的语句，除非很有必要时。(1.42+)

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于设计与算法。

8. 明确方法功能，精确（而不是近似）地实现方法设计。一个函数仅完成一件功能，即使简单功能也编写方法实现。(1.42+)

说明：虽然为仅用一两行就可完成的功能去编方法好象没有必要，但用方法可使功能明确化，增加程序可读性，亦可方便维护、测试。

- 9. 应明确规定对接口方法参数的合法性检查应由方法的调用者负责还是由接口方法本身负责，缺省是由方法调用者负责。(1.42+)**

说明：对于模块间接口方法的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

- 10. 尽量使用Java 5.0新循环写法。(5.0+)**

说明：代码更加简洁

示例：

```
ArrayList<String> list = new ArrayList<String>();
list.add...
for(String str:list)
{
    System.out.println(str);
}
```

- 11. 使用Java 5.0枚举来替代以前用数字与字符串的同等目的的操作。(5.0+)**

说明：Java 5.0以前没有枚举，大家都用数字或者字符串做枚举同样功能的事情

示例：

```
public enum EnumDemo
{
    ERROR, INFO, DEBUG
}

In other function:
EnumDemo t = EnumDemo.DEBUG;
if (t == EnumDemo.ERROR)
{
    . . . . .
}
```

- 12. interface 中定义的常量不要写public、static、final的修饰词，方法不要写public修饰词。(1.42+)**

说明：更加简洁

示例：

```
public interface InterfaceT
{
    String TT = "abcd";
    void doStart();
}
```

- 13. 新起一个线程，都要使用Thread.setName("...")设置线程名。**

说明：性能测试时可对线程状态进行监控，异常时也可以知道异常发生在哪个线程中

1.5. 性能与可靠性

1.5.1. 规则

1. 对Debug，Info级别日志输出前必须对当前的调试等级先进行判断。(1.42+)

说明：日志一般都会有不少字符串的处理，如果不是Debug级别就没有必要进行处理

示例：

```
if (logger.isDebugEnabled())
{
    logger.debug("request : " + request.getMethod());
}
```

2. 数组复制使用System.arraycopy(*)。(1.42+)

说明：更好的性能

3. 不要使用循环将集合转为数组，可以使用集合的toArray()方法。(1.42+)

说明：更好的性能，代码更加简洁

示例：

```
ArrayList list = new ArrayList();
list.add....
String [] array = new String[list.size()];
list.toArray(array);
```

4. 大量字符串的相加等于处理应该使用StringBuffer。(1.42+)

说明：大量的String相加等于处理性能消耗较多。“大量”一般指5次“+=”以上或者在循环中进行字符串+=操作。

示例：

不推荐：

```
String str = "";
str += "a";
str += "b";
```

推荐：

```
StringBuffer sb = new StringBuffer();
sb.append("aa");
```

```
sb.append("bb");
sb.append("cc");
```

5. 对类中日志工具对象logger应声明为static. (1.42+)

说明：防止重复new出logger对象(logger指各种日志工具类，统一使用log4j或内部API进行日志打印，尽管一些logger对LogFactory工厂有一些优化，但是我们也必须防止代码没有必要的运行)。

1.5.2. 建议

1. public类型的底层函数需对输入参数进行判断，参数不合法应该主动抛出RuntimeException. (1.42+)

说明：底层函数必须保证输入参数正确性再进行其他处理(防止后面的代码抛出错误，减少没有必要的后续代码运行)。使用RuntimeException 减少了try catch满天飞，并有利于快速定于异常代码。

示例：

```
public void doDivide(int a,int b)
{
    if (b == 0)
    {
        throw new IllegalArgumentException("denominator can't be
        zero");
    }
    ...
}
```

2. 尽量使用JDK自带的API函数，不要自己写类似功能的函数. (1.42+)

说明：JDK自身的函数在可靠性，性能方面一般有更好的表现，大家必须熟练掌握，特别是算法方面的。

3. IO操作流使用有Buffer功能的class. (1.42+)

说明：更好的性能，没有Buffer的输出流频繁IO操作，效率反倒低。

示例：

```
FileOutputStream file= new FileOutputStream("test.txt");
BufferedOutputStream out = new BufferedOutputStream(file);
for (int i = 0; i < bytes.length; i++ )
{
    out.write(...);
}
```

```

    }
    out.flush();

```

1.6. 接口

1.6.1. 规则

1. 接口注释务必详尽，包括参数和返回值的说明

说明：不要把问题留给调用接口的同事，接口注释尽可能详尽。

2. 升级或变更接口时不要删除原有的接口

说明：升级或变更接口时，首先保留原有接口，不要伤害调用你接口的同事。然后在原有接口加上注释关键字 `@deprecated`，声明该接口已经被弃用。

1.7. Gitlab

1.7.1. 规则

1. 修改代码前更新代码

说明：修改代码前先更新代码，避免出现代码版本混乱。

2. 提交代码请清楚说明提交的更新

说明：提交代码请写明提交更新的模块及详细内容，一方面方便其他协作的同事了解提交的版本是否影响其coding模块，另一方面是为了支持后期版本管理。

2. 附录

建议在开发工具中使用以下模板：

2.1. Eclipse 知微风格



Zhiwei style for eclipse.xml

2.2. Eclipse 注释模板



comment_template_zhiwei.xml

2.3. JBuilder风格(Jbuilder2007以下版本,不包括2007版)



zhiwei.codestyle